AFRL-RI-RS-TR-2018-016

VINE: A VARIATIONAL INFERENCE-BASED BAYESIAN NEURAL NETWORK ENGINE

UNIVERSITY OF SOUTHERN CALIFORNIA

JANUARY 2018

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

/ S /	**/ S /**
CHRISTOPHER J. FLYNN	JOHN D. MATYJAS
Work Unit Manager	Technical Advisor, Computing & Communications Division
	Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* JAN 2018	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* OCT 2016 – AUG 2017

4. TITLE AND SUBTITLE VINE: A VARIATIONAL INFERENCE-BASED BAYESIAN NEURAL NETWORK ENGINE	**5a. CONTRACT NUMBER** N/A
	5b. GRANT NUMBER FA8750-17-2-0021
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Massoud Pedram, Yanzhi Wang	**5d. PROJECT NUMBER** SAGA
	5e. TASK NUMBER US
	5f. WORK UNIT NUMBER C1
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Southern California 3720 S. Flower St, CUB 303, MC 0701 Los Angeles CA 90089-0701	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2018-016

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes our findings and results for the DARPA MTO seedling project titled “SpiNN-SC: Stochastic Computing-Based Realization of Spiking Neural Networks” also known as “VINE: A Variational Inference-Based Bayesian Neural Network Engine.” The primary goal was to develop a Bayesian Neural Network (BNN) with an integrated Variational Inference (VI) engine to perform inference and learning (statically and on-the-fly) under uncertain or incomplete input and output features. A secondary goal is to enable robust decision making under noise and variability in the observed data and without reference to a ground truth. The key expected impact is to enable a new generation of BNNs that can operate on input and output features specified as random variables, that admit efficient hardware realization, and that can not only do inference but also can be retrained on-the-fly based on incoming data.

15. SUBJECT TERMS
Machine learning, Neural networks, Inference engine, Independent component analysis, Hardware random number generation, Transfer learning

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** CHRISTOPHER J. FLYNN
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	26	**19b. TELEPHONE NUMBER** (*Include area code)*

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

List of Figures

List of Tables

1. Summary

This report describes our findings and results for the DARPA MTO seedling project titled "SpiNN-SC: Stochastic Computing-Based Realization of Spiking Neural Networks" also known as "VINE: A Variational Inference-Based Bayesian Neural Network Engine." The report is accompanied by the full set of prototype software and hardware design deliverables for the said project.

2. Introduction

The primary goal was to develop a Bayesian Neural Network (BNN) with an integrated Variational Inference (VI) engine to perform inference and learning (statically and on-the-fly) under uncertain or incomplete input and output features. A secondary goal is to enable robust decision making under noise and variability in the observed data and without reference to a ground truth. The key expected impact is to enable a new generation of BNNs that can operate on input and output features specified as random variables, that admit efficient hardware realization, and that can not only do inference but also can be retrained on-the-fly based on incoming data.

2.1 Description of the Technical Approach

The approach comprised of the following steps (see Figure 1).

- Modeling of input and output features as random variables with arbitrary probability density functions (pdfs).
- Cardinality reduction of the input feature space based on a type of independent component analysis (ICA).
- Efficient generation of random values adhering to a desired arbitrary (but monotonic) pdf.
- Implementation of a BNN with integrated VI engine and the ability to accept, process and store random variables as inputs or outputs.
- Analysis of target application data and data/environmental modeling.
- System integration and demonstration including hardware and software prototyping.

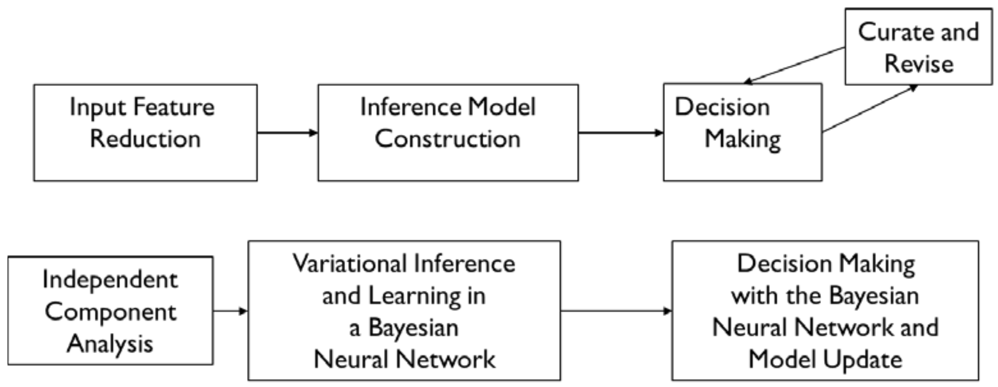

Curate and Revise
Input Feature Reduction
Inference Model Construction
Decision Making
Independent Component Analysis
Variational Inference and Learning in a Bayesian Neural Network
Decision Making with the Bayesian Neural Network and Model Update

Figure 1: Overall Flow of the VINE

3. Methods, Assumptions, and Procedures

The project team developed software and prototype hardware realization of a variational inference engine for Bayesian neural network driven by the target application requirements. In the rest of this section, we first explain the VIBNN construction, training, and inference engine followed by the knowledge transfer framework of a variational inference-based Bayesian neural network (VIBNN) which can operate on training data without a ground truth. This is followed by descriptions of the random number generator and input dimension reduction modules.

3.1 Construction and Optimization of the VIBNN

3.1.1 VIBNN Construction, Training, and Inference Engine

As shown in Figure 2, a BNN has a set of latent variables, $\mathbf{z} = \{z_1, z_2, \dots\}$, which includes edge weights w_{ij}^{l} from the i-th neuron at the layer l to the j-th neuron at the layer $l+1$ and biases $\mathbf{b}^{l}$ at the layer l. Different from conventional artificial neural networks (ANNs), the latent variables in a BNN is specified as distributions instead of scalar values. Equivalently, a BNN can be viewed as an ensemble of a large number of neural networks with introduced variabilities on weights and biases. As a result, BNNs are robust to disturbances in the learning especially when the training set is noisy or incomplete. In addition, the over-fitting problem can be alleviated because regularizations are implicitly introduced by imposing prior distributions that are not uniform.

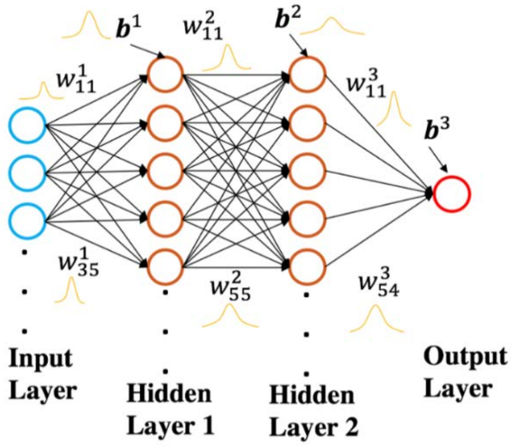


Figure 2: Structure of a VIBNN

The goal of learning a BNN is to find the posterior distribution over its latent variables $\boldsymbol{z}$ given a training dataset $\mathcal{D}$. As we know from the Bayes' rule, the posterior distribution of latent variables, comprising of all node biases and edge weights in the network, denoted by $p(\boldsymbol{z}|\mathcal{D})$, can be calculated as follows

$$p(\boldsymbol{z}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{z}) \cdot p(\boldsymbol{z})}{\int_{\boldsymbol{z}'} p(\mathcal{D}|\boldsymbol{z}')\mathrm{d}\boldsymbol{z}'}$$

where $p(\boldsymbol{z})$ is the prior distribution that is assumed *a priori*, and $p(\mathcal{D}|\boldsymbol{z})$ is the likelihood function which can be calculated based on the output of the network. However, for most cases of interest, the integral in the denominator is intractable, which calls for efficient approximations. One of the most commonly used methods is the Markov Chain Monte Carlo (MCMC), which simulates a Markov chain whose stationary (invariant) states follow a given (target) probability distribution in a very high dimensional state space and which goal is to generate "fair" samples of the said state space by trying to identify and sample high probability states. However, MCMC is computationally very expensive and lacks a clear stopping criterion.

We adopt a learning and inference method called *Bayes by Backprop* as proposed in [1]. To efficiently learn the distributions of the latent variables, the idea of variational inference is adopted in which the posterior distribution of latent variables given a dataset $\mathcal{D}$, denoted by $p(\boldsymbol{z}|\mathcal{D})$, is approximated using a distribution $q(\boldsymbol{z};\boldsymbol{\theta})$ with a fixed form but a set of unknown parameters $\boldsymbol{\theta}$. To learn the value of $\boldsymbol{\theta}$, one can minimize the Kullback-Leibler (KL) divergence between the variational posterior $q(\boldsymbol{z};\boldsymbol{\theta})$ and the actual posterior $p(\boldsymbol{z}|\mathcal{D})$. More precisely, finding the best estimation of $\boldsymbol{\theta}$, denoted by $\boldsymbol{\theta}^*$, is an optimization problem expressed as follows

$$\begin{aligned}\boldsymbol{\theta}^* &= \underset{\theta}{\operatorname{argmin}} \int_{-\infty}^{+\infty} q(\boldsymbol{z};\boldsymbol{\theta}) \log \frac{q(\boldsymbol{z};\boldsymbol{\theta})}{p(\boldsymbol{z}|\mathcal{D})} \mathrm{d}\boldsymbol{z} \\ &= \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{q(\boldsymbol{z};\boldsymbol{\theta})}\left[\log \frac{q(\boldsymbol{z};\boldsymbol{\theta})}{p(\boldsymbol{z}|\mathcal{D})}\right] \\ &= \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{q(\boldsymbol{z};\boldsymbol{\theta})}[\log q(\boldsymbol{z};\boldsymbol{\theta}) - \log p(\mathcal{D}|\boldsymbol{z}) - \log p(\boldsymbol{z})]\end{aligned}$$

where the expectation is often evaluated using a Monte Carlo integral by taking a small number of sample points according to $q(\boldsymbol{z};\boldsymbol{\theta})$. For convenience of expression, we define $\mathcal{F}(\boldsymbol{\theta})$ as the objective function on the right-hand side of the equation above. In practice, the optimization problem can be solved using the gradient descent method.

In our case, $q(\boldsymbol{z};\boldsymbol{\theta})$ is assumed to be Gaussian. The set of parameters is $\boldsymbol{\theta} = \{\boldsymbol{\mu}, \Sigma\}$, where $\boldsymbol{\mu}$ is the mean vector, and $\Sigma = \operatorname{diag}(\sigma_1^2, \sigma_2^2, \ldots)$ is a diagonal covariance matrix. For simplicity, we define $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots)$As a result, the latent variables can be re-parameterized in terms of $\boldsymbol{\theta}$ as follows

$$\begin{aligned}\boldsymbol{z} &= \boldsymbol{\mu} + \Sigma\boldsymbol{\epsilon} \\ &= \boldsymbol{\mu} + \boldsymbol{\sigma} \circ \boldsymbol{\epsilon}\end{aligned}$$

where "∘" is element-wise multiplication operation, and $\boldsymbol{\epsilon} \sim N(\boldsymbol{0}, \boldsymbol{I})$. Furthermore, to eliminate the constraint that $\boldsymbol{\sigma}$ is non-negative, we introduce $\boldsymbol{\rho} \in \mathbb{R}$ such that $\boldsymbol{\sigma} = \log(1 + \exp \boldsymbol{\rho})$. Since there is a one-to-one mapping from $\boldsymbol{\rho}$ to $\boldsymbol{\sigma}$, we will optimize $\boldsymbol{\rho}$ instead of $\boldsymbol{\sigma}$. Consequently, $\mathcal{F}(\boldsymbol{\theta})$ can be rewritten as

$$\mathcal{F}(\boldsymbol{\theta}) = \mathbb{E}_{q(\boldsymbol{\epsilon})}[\log q(\boldsymbol{z};\boldsymbol{\mu},\boldsymbol{\rho}) - \log p(\mathcal{D}|\boldsymbol{z}) - \log p(\boldsymbol{z})]$$

And the partial derivative of $\mathcal{F}(\boldsymbol{\theta})$ with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\rho}$ can in turn be written as

$$\frac{\partial \mathcal{F}(\boldsymbol{\theta})}{\partial \boldsymbol{\mu}} = \mathbb{E}_{q(\boldsymbol{\epsilon})}\left[\frac{\partial f(\boldsymbol{z};\boldsymbol{\theta})}{\partial \boldsymbol{z}} + \frac{\partial q(\boldsymbol{z};\boldsymbol{\mu},\boldsymbol{\rho})}{\partial \boldsymbol{\mu}}\right]$$
$$\frac{\partial \mathcal{F}(\boldsymbol{\theta})}{\partial \boldsymbol{\rho}} = \mathbb{E}_{q(\boldsymbol{\epsilon})}\left[\frac{\partial f(\boldsymbol{z};\boldsymbol{\theta})}{\partial \boldsymbol{z}} \circ \frac{\boldsymbol{\epsilon}}{1+\exp(-\boldsymbol{\rho})} + \frac{\partial q(\boldsymbol{z};\boldsymbol{\mu},\boldsymbol{\rho})}{\partial \boldsymbol{\rho}}\right]$$

where

$$f(\boldsymbol{z};\boldsymbol{\theta}) = \log q(\boldsymbol{z};\boldsymbol{\theta}) - \log p(\mathcal{D}|\boldsymbol{z}) - \log p(\boldsymbol{z})$$

With $q(\boldsymbol{z};\boldsymbol{\theta})$ local to each parameter (for a diagonal covariance matrix), $p(\mathcal{D}|\boldsymbol{z})$ available at the network output after a round of forward propagation, and $p(\boldsymbol{z})$ assumed as prior knowledge (Gaussian in our case), the value of $\frac{\partial f(\boldsymbol{z};\boldsymbol{\theta})}{\partial \boldsymbol{z}}$ can be found using standard back-propagation techniques from the output layer to the input layer. Meanwhile, $\frac{\partial q(\boldsymbol{z};\boldsymbol{\mu},\boldsymbol{\rho})}{\partial \boldsymbol{\mu}}$ and $\frac{\partial q(\boldsymbol{z};\boldsymbol{\mu},\boldsymbol{\rho})}{\partial \boldsymbol{\rho}}$ can be directly computed independent of the network structure. After the gradient is obtained, gradient descent can be applied as

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{F}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

where α is the learning factor. We will refer to the BNN learned via variational inference as explained above as the VIBNN. Considering the high computational complexity of learning a BNN due to integral computation using Monte Carlo sampling, we choose not to include the learning algorithm in the proposed FPGA realization. Instead, we choose to learn the BNN in software and load the learned network parameters into the FPGA platform for inference.

One simple way to use the learned BNN for inference is to put the *maximum a posteiori* (MAP) estimation of the latent variables in the place of the edge weights and node biases of an ANN. This way, any existing inference framework designed for an ANN can also be applied to a BNN. However, to fully utilize the posterior distribution of the network weights which better represent the network, one should derive the network output by averaging over the output produced by the ensemble of networks specified according to the posterior distribution. In other words, instead of using a point estimation of the latent variables, one should sample the latent variables multiple times according to the posterior distribution, each producing a set of outputs, and then averaging over all the produced output. Formally, given the input features $\boldsymbol{x}_0$, if the network function is denoted by $\boldsymbol{y} = g(\boldsymbol{x}_0;\boldsymbol{z})$ where $\boldsymbol{y}$ is the output of the network after one round of forward propagation using latent variables (edge weights and node biases) $\boldsymbol{z}$, then the output of the VIBNN, denoted by $\hat{\boldsymbol{y}}$, given $\boldsymbol{x}_0$ and $q(\boldsymbol{z};\boldsymbol{\theta})$ is computed as

$$\begin{aligned}\hat{\boldsymbol{y}} &= \mathbb{E}_{q(\boldsymbol{z};\boldsymbol{\theta})}[g(\boldsymbol{x}_0;\boldsymbol{z})] \\ &\cong \sum_{s=1}^{M} g(\boldsymbol{x}_0;\boldsymbol{z}_s)\end{aligned}$$

where $\boldsymbol{z}_s$ is a sample of $\boldsymbol{z}$ according to $q(\boldsymbol{z};\boldsymbol{\theta})$, and M is the number of samples used (typically set to 5 or 10).

3.1.2 Knowledge Transfer from Outcome Prediction to Action Recommendation

A conventional BNN cannot be used for action recommendation without encountering the “no ground truth” problem. Therefore, we propose a knowledge transfer framework which first constructs a BNN for outcome prediction only (i.e. latent model construction) and then transfer the knowledge learned to the domain of action recommendation. A graphical demonstration of the proposed knowledge transfer framework can be found in Figure 3 and Figure 4.

In the latent model construction phase, the BNN for outcome prediction is learned by only considering the actions reported in the training dataset. Therefore, standard learning techniques such as a VIBNN can be used for this phase. Note that we consider both “good” actions and “bad” actions in this phase in the hope that the latent model network can adapt to a wide range of input cases.

In the knowledge transfer phase, another BNN is trained with recommended actions as outputs. Since the reported actions are not guaranteed to be the best possible ones, only inputs of the samples in the training dataset are used so that the inference engine is trained with the right distribution over inputs. While the output values (i.e. desirable actions) for the inference engine cannot be explicitly set during the training phase (because there are no assumed ground truths), desirable actions are implicitly specified as those that produce desirable outcomes (which can be deduced from pre-existing domain knowledge.) For instance, in the context of an emergency room visit by a patient, although we cannot easily know what prescription to give to a patient, our goal is to keep the patient’s vital signs within a healthy range. Therefore, by concatenating the latent model network after the inference engine (which is yet to be learned) and placing the desirable outcome for each input sample as the final output, we can guide the inference engine to produce better actions. The loss function required for learning a VIBNN can then be expressed by a measure of difference between the desirable outcome and the (estimated) achieved outcome with the current inference engine (e.g., mean squared error or cross entropy.) Note that while the network parameters in the latent model should not be modified anymore in this training phase, the underlying BNN structure is amenable to updates by the neural back-propagation algorithm in combination with any with any gradient-based optimizer such as the gradient descent method. Clearly, the back-propagation algorithm plus the gradient descent method can be employed to learn the latent variables of the inference engine.

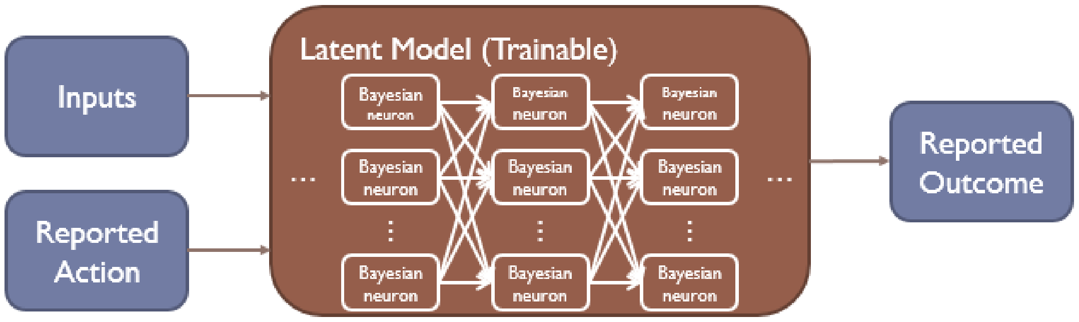


Figure 3: Step 1: latent model construction

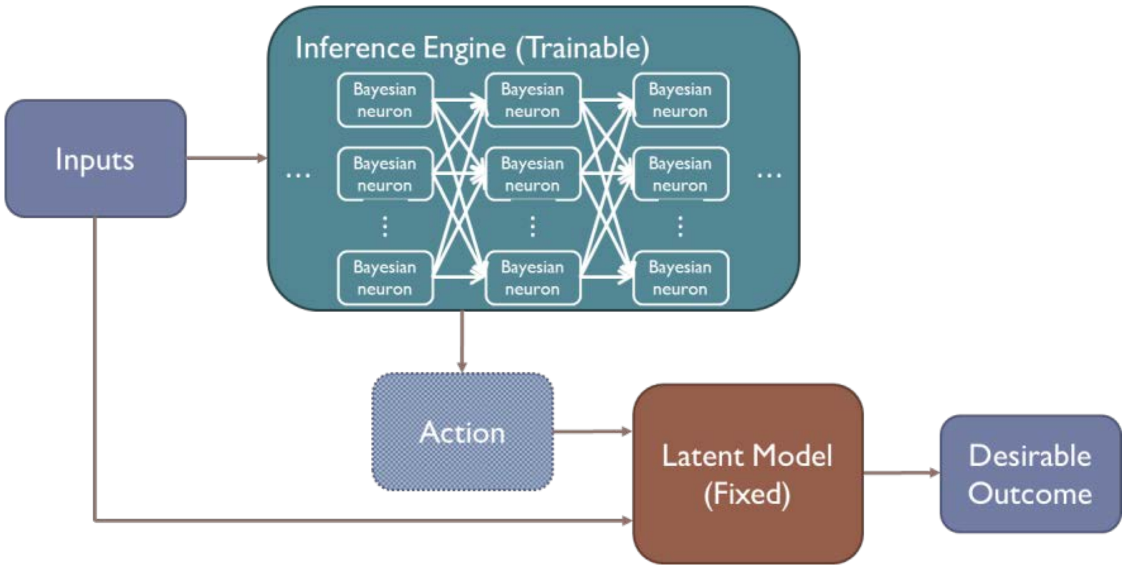


Figure 4: Step 2: knowledge transfer

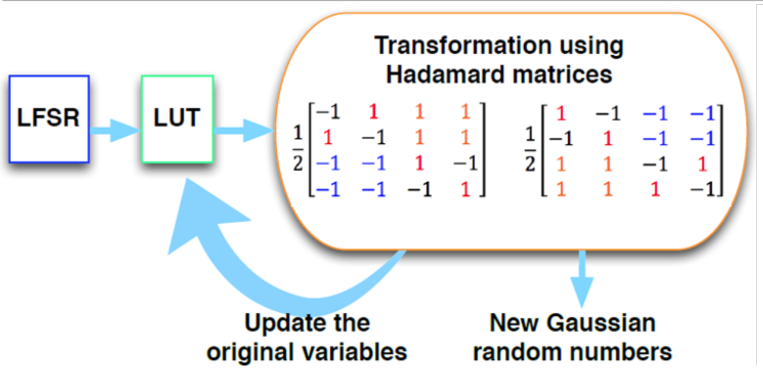
3.1.3 Random Number Generator Design

To implement a VIBNN in hardware, one crucial step is to design a Gaussian random number generator (GRNG) that has high bandwidth and low resource usage.

Our first attempt is a GRNG using the Wallace algorithm (as shown in Figure 5). After a lookup table of $K \times L$ Gaussian random numbers are initialized as “seeds”, the Wallace algorithm can generate a Gaussian random number flow by (i) sampling K elements from the lookup table at random (using an LFSR or a Tausworth RNG), (ii) performing a linear transformation on the K samples using a Hadamard matrix, (iii) updating the lookup table with transformed values, and (iv) returning the transformed values as outputs once in R iterations. A block diagram of our implementation is shown in Figure 6. Our preliminary results show that a small size of lookup table of 128 or 256 entries with a 16-bit fixed point number for each entry will be sufficient to generate random numbers for the Bayesian neuron, and random numbers could be generated in each clock cycle.

Later on, we observe that the Wallace GRNG has a large memory overhead and design a *RAM-based Linear Feedback Gaussian Random Number Generator* (RLF-GRNF) which is ideal for parallel random number generation.

```
1:  for $i = 1..R$ do   {R = retention factor}
2:    for $j = 1..L$ do   {L = N/K}
3:      for $z = 1..K$ do   {K = matrix size}
4:        $x[z] \leftarrow pool[\text{generate\_addr}()]$
5:      end for      {Apply matrix transformation to the K values}
6:      $x' \leftarrow \text{transform}(x)$
7:      for $z = 1..K$ do   {write K values to pool}
8:        $pool[\text{generate\_addr}()] \leftarrow x[z]'$
9:      end for
10:   end for
11: end for
12: $S \leftarrow \sqrt{pool[N]/N}$   {Approximate a $\chi^2_N$ correction for sum of squares.}
13: return $pool[1..(N-1)] \times S$   {Return pool with scaled sum of squares.}
```



LFSR
LUT
Transformation using Hadamard matrices
$\frac{1}{2}\begin{bmatrix}-1 & 1 & 1 & 1\\ 1 & -1 & 1 & 1\\ -1 & -1 & 1 & -1\\ -1 & -1 & -1 & 1\end{bmatrix}$
$\frac{1}{2}\begin{bmatrix}1 & -1 & -1 & -1\\ -1 & 1 & -1 & -1\\ 1 & 1 & -1 & 1\\ 1 & 1 & 1 & -1\end{bmatrix}$
Update the original variables
New Gaussian random numbers

Figure 5: Wallace algorithm

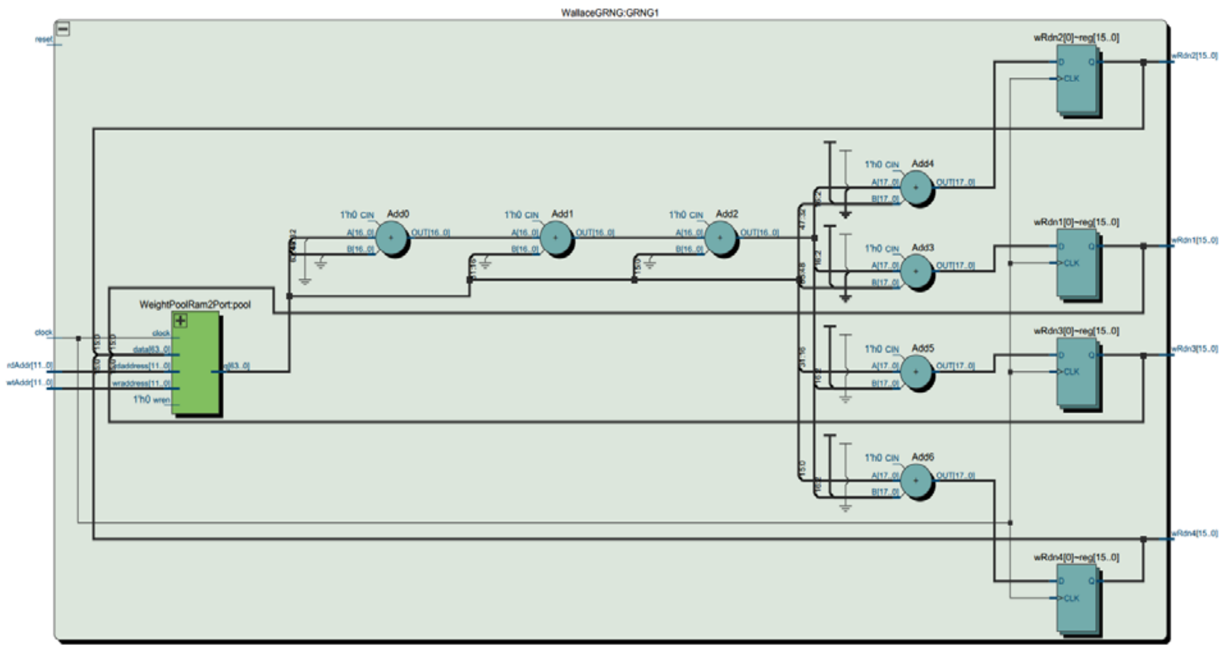


Figure 6: Block diagram of a Wallace GRNG

Precisely, the binomial approximation method [2] has been adopted in the design of a Gaussian random number generator. This method is implemented by using a 128-bit Linear Feedback Shift Register (LFSR) for parallel random seed bit production and a Parallel Counter (PC) to convert the number of 1's in the LFSR to a binary number. An LFSR is capable of exhaustive pseudo-random number generation and is easy to implement in hardware. The PC can be implemented using adders in a tree structure. This method is, however, not suitable for parallel implementation due to its high usage of LFSR registers. That is why we opted to implement the linear feedback function in a RAM as described below.

The architecture of a RAM-based linear feedback GRNG (RLF-GRNG) is shown in Figure 7. The RAM block is used for seed bits storing. Instead of performing shifting as LFSRs, the RLG-GRNG method uses an indexer to perform the equivalent operation in RAM. The indexer memorizes the head and tap locations, and increments their values in each iteration.

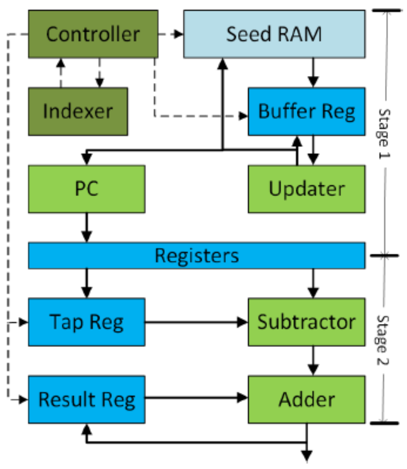


Figure 7: Control and Data Flow Graph of an RLF-GRNG

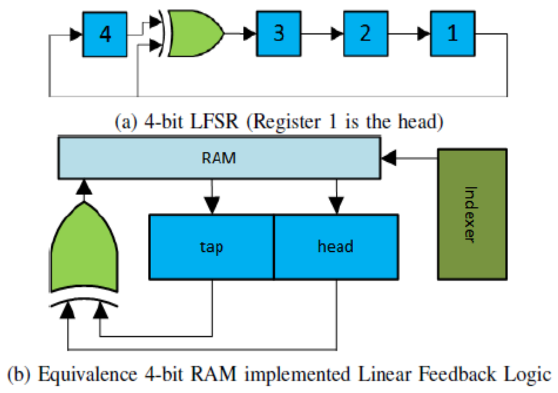

4
3
2
1
(a) 4-bit LFSR (Register 1 is the head)
RAM
tap
head
Indexer
(b) Equivalence 4-bit RAM implemented Linear Feedback Logic

Figure 8: Comparison of register- and RAM-based LSFRs

As a simple example shown in Figure 8, the LFSR uses fixed head locations in which the head is always the MSB or the LSB of the register, and shift its contents in each cycle. The RAM based linear feedback logic stores all seeds in a RAM block and uses an indexer to track the head location and the taps locations. The buffer register stores the values of the current taps and head. In addition, the buffer register and the updater shifts its updated values to minimize the RAM access frequency if there are taps that are consecutive (in other words, a tap whose value still requires updating in the following iteration can be stored in the buffer register instead of being written back to the RAM in the current iteration and being read again in the following iteration. The updater performs the XOR function with the head for each tap and updates their values. The controller implements a state machine. In each state, the head or one of the taps is updated. The PC calculates the number of 1's in the new taps values, while the previous values are stored in the tap register. The rest of the operations are performed in the second stage. The tap difference is obtained using the subtractor before being added to the previous result which is stored in the result register. The initial result values can be pre-calculated and stored in a ROM block for the result register initialization.

3.1.4 Input Dimension Reduction

Optionally, an input dimension reduction module can be included in the design. We choose to implement an algorithm using the independent component analysis (ICA) method.

In the standard linear model, input features are modeled as linear combinations of some independent components:

$$x_{m\times 1} = A_{m\times n} s_{n\times 1} \quad m \geq n$$

where x is a column vector of input features, A is the mixing matrix comprised of row vectors $a_i, i = 1,2,\ldots,m$, s is a column vector of random independent components $s_j, j = 1,2,\ldots,n$, m is the dimensionality of input features, and n is the dimensionality of independent components. Independent components are assumed to be non-stationary, so that different linear models may be in effect at different times.

The objective of ICA is to find a separation matrix that finds an estimate of independent components, without having any prior information about independent components, s, or the mixing matrix, A. This can be written as:

$$y_{n\times 1} = B_{n\times m} x_{m\times 1}$$

where y is a column vector of estimates of independent components and B is the separation matrix.

One of the major advantages of ICA over other dimensionality reduction techniques such as PCA and factor analysis is that it deals with non-Gaussian distributions, e.g. heavy-tailed distributions that are common in many real-world datasets. Another advantage of ICA is that it finds components that are statistically independent. This property has an impact on machine learning models that deal with probability density functions (PDFs). For example, in Bayesian neural networks where inputs, weights, and/or outputs are represented by PDFs, a challenging and computationally expensive step is sampling these possibly dependent density functions. This problem becomes more complicated when the dependency among distributions involves higher-order statistics (HOS). Consequently, if ICA is applied to input features as a preprocessing step, the PDF of each feature in reduced space can be easily sampled independent of other features.

There are two general ways for estimating independent components. The first one is by direct use of HOS and by maximizing a measure of non-Gaussianity. The intuition behind these methods is that because sum of two random variables is closer to a Gaussian than original ones, estimated components are independent when a measure of non-Gaussianity is maximized. The second one is by indirect use of HOS through nonlinear decorrelation. The rationale behind these methods is that, if y_i and y_j are independent, any nonlinear transformations $g(y_i)$ and $h(y_j)$ are uncorrelated. Therefore, they try to find the separation matrix such that y_i and y_j are uncorrelated and transformed components $g(y_i)$ and $h(y_j)$ are also uncorrelated.

Equivariant Adaptive Separation via Independence (EASI) [3] is a gradient-based algorithm that estimates independent components using nonlinear decorrelation. EASI has several advantages compared to other algorithms for ICA. First, it is an adaptive algorithm which makes it suitable for problems where underlying distributions of input features change. In problems where adaptivity is not a must, there are superior algorithms such as FastICA, which seeks an orthogonal rotation of whitened data through a fixed-point iteration scheme. Second, it is equivariant, i.e. convergence rates, stability conditions, and interference rejection levels depend only on distributions of source signals and are independent of the mixing matrix. Third, unlike other methods that require whitening of input features as a preprocessing step, it merges whitening with separation, which improves parallelism. Last but not least, the basic operations are computationally efficient since it only requires addition and multiplication.

Figure 9 shows an overview of the EASI algorithm and the operations required to implement it. First, the separation matrix is initialized with random values. Then, in each iteration, the separation matrix is multiplied by input features to generate output features. A nonlinear function $g(.)$ is applied element-wise to output features to introduce HOS to the problem. The output of nonlinear function and output features are fed to the module that calculates relative gradient H (aka natural gradient). Finally, relative gradient is multiplied by learning rate μ to update elements of the separation matrix for the next iteration. The same steps are repeated until convergence.

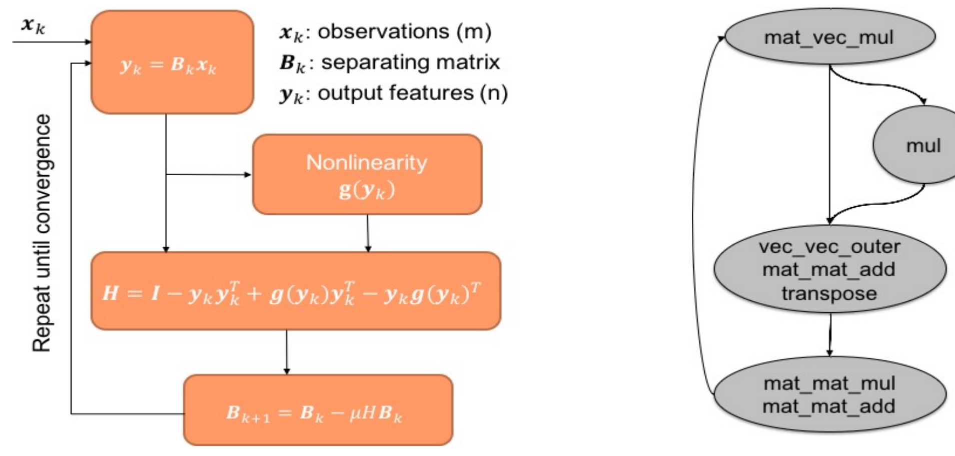


Figure 9: Block diagram of EASI and required operations

4. Results and Discussions

We delivered the following reports, software packages, and hardware as part of this project.

Software packages and hardware deliverable:

- Hardware realization of a random number generator meeting an arbitrary monotonic pdf (based on the mean field sampling framework) – delivered in April 2017. GRNG.zip:

Verilog code for Wallace GRNG and RLF-GRNG in the attached source code and data files.

- Implementation of the EASI Algorithm for input feature reduction (software and hardware) – delivered in May 2017. See EASI.zip: Software and hardware code for input feature reduction module in the attached source code and data files
- Implementation of the VIBNN for some target applications (software and hardware) – delivered with this report. See VIBNN.zip: Software (Tensorflow) and hardware (Verilog) code for VIBNN for different applications in the attached source code and data files.
- System integration, test and performance evaluation – delivered as part of this report. See evaluation and results below.

The aforesaid software and hardware design files can be downloaded from the following password protected URL: http://sportlab.usc.edu/downloads/download-protected/. For username and password information, please write to PI, Massoud Pedram, at pedram@usc.edu.

Reports and publications:

1. M. Nazemi, S. Nazarian and M. Pedram, "High-performance FPGA implementation of equivariant adaptive separation via independence algorithm for Independent Component Analysis," in Proceedings of the 28th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2017.
2. R. Cai, A. Ren, L. Wang, M. Pedram and Y. Wang, "Hardware Acceleration of Bayesian Neural Networks using RAM based Linear Feedback Gaussian Random Number Generators," in Proceedings of the IEEE International Conference on Computer Design (ICCD), 2017.
3. R. Cai, A. Ren, N. Liu, L. Wang, M. Pedram, and Y. Wang. “VIBNN: Hardware Acceleration of Bayesian Neural Networks,” to appear in Proc. of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Mar. 2018.

4.1 Evaluation of the Knowledge Transfer Framework in VINE

To evaluate the performance of the proposed knowledge transfer framework, we use the UCI mushroom dataset [4], which contains descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species is identified as edible or poisonous. There is no simple rule for determining the edibility of a mushroom. It contains 8,124 instances, each consisting of 22 features (e.g., cap-shape, stalk-shape, habitat, etc.). The whole dataset is split of which 80% is used for training and 20% is used for testing. Since all features are of enumeration type, we apply one-hot encoding and get a total of 118 binary features as a result. Binary actions (whether to eat a mushroom or not) are considered while non-deterministic rewards are assigned based on the edibility of the mushroom and the action taken. When one chose not to eat, the reward is set to zero. If a poisonous mushroom is eaten, a reward is generated from a Gaussian distribution $N(-3.0, 2.25)$. On the other hand, if an edible mushroom is eaten, a reward is generated from a Gaussian distribution $N(3.0, 2.25)$. The goal is to maximize the average reward given the features of mushrooms in the test set. While actions are not included in the original data, we preprocess the dataset to generate reported actions. It is assumed that 80% of reported actions are correct (i.e. to eat an edible mushroom or not to eat a poisonous mushroom).

The BNN used for latent model construction (referred to as BNN IAO) generates predicted outcome in terms of rewards while taking the input features and reported actions. The BNN used for action recommendation (referred to as BNN IA) produces possible best actions in terms of possibilities from a *softmax* normalization from input features and the output of the BNN IAO. In Table 1, the performance of the proposed inference engine with various sizes of hidden layers is compared against three baselines, namely, (i) "always eat", (ii) "never eat", and (iii) "take the reported action". The mean squared error (MSE) between rewards produced by the trained latent model BNN IAO and rewards for the training dataset at the training epoch 10 decreases from 1.1827 to 1.1793. This indicates BNN with smaller network size would be more accurate for reward prediction after a same number of training epochs. The mean reward achieved by taking recommended actions produced form BNN IA would decrease from 1.5569 to 1.5233 when the hidden layer size reduces from 200 to 32. Except for the as-mentioned difference in network size, all networks are trained using the same dataset and hyper parameter settings as discussed.

Table 1 Performance evaluation of the proposed transfer learning framework at different network configurations

BNN IAO Configuration	BNN IA Configuration	Mean Reward (BNN IA)	Mean Reward (always eat)	Mean Reward (reported)	Mean Reward (oracle)	BNN IAO (MSE)
119x200x200x1	118x200x200x1	1.5569	0.1104	0.9596	1.5622	1.1827
119x128x128x1	118x128x128x1	1.5501	0.1218	0.9596	1.5692	1.1846
119x64x64x1	118x64x64x1	1.5306	0.1416	0.9596	1.5621	1.1750
119x32x32x1	118x32x32x1	1.5233	0.1305	0.9596	1.5827	1.1793

4.2 Hardware Implementation of the VINE

4.2.1 Random Number Generator Design

The block diagram of the implemented RLF-GRNG is shown in Figure 10. The Initialization ROM stores the initial summation results of the seeds RAM in the GRNG. The seeds RAM stores all seeds for random variable generation. Each bit of the word read from the RAM is propagated to an LF-updater for tap update and random variable calculation. Each LF-updater implements the 2-stage pipeline structure discussed above. The updated taps per tap location are collected from all LF-updaters and formed into one word to be written back to the seed RAM. The results generated by every four LF-updaters are selected sequentially by 4 outputs with different orders through its multiplexer for enhanced randomness. All select signals are shared and generated by the controller. The controller also produces indices and memory access signals for the seeds RAM, as well as command signals for all LF-updaters.

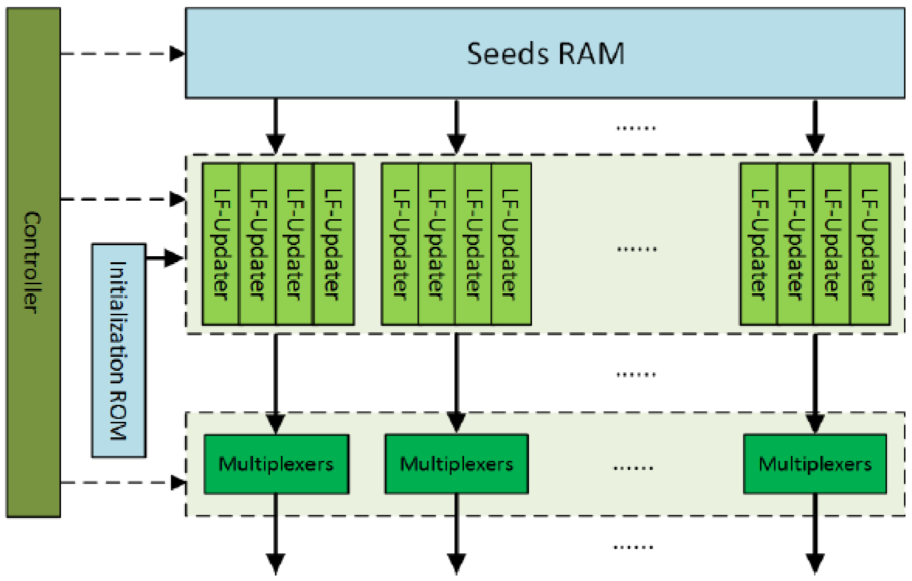


Figure 10: Block diagram of the RLF-GRNG

A comparison of hardware utilization of Wallace GRNG and RLF-GRNG to generate 64 random numbers in parallel is shown in Table 2. One can see that the RLF-GRNG implementation achieves significant saving on the amount of required memory.

Table 2 Summary of hardware resource utilization

	Wallace GRNG	RLF-GRNG
ALM	401/113,560 (< 1%)	831/113,560 (< 1%)
Register count	1166	1780
Block memory bits	1,048,576/12,492,800 (8%)	16,384/12,492,800 (< 1%)
RAM blocks	103/1220 (8%)	3/1220 (< 1%)

4.2.2 Input Dimension Reduction

Figure 11 provides an overview of the hardware realization flow of the EASI algorithm. More precisely, to realize EASI in hardware, the following steps are taken. The algorithm is implemented in software to tune the learning rate and find an appropriate nonlinear function. Design space exploration follows to investigate the power-performance trade-offs of various architectures. Finally, the target circuit is implemented in Chisel [5] for hardware realization. The details of each step are explained later in the report.

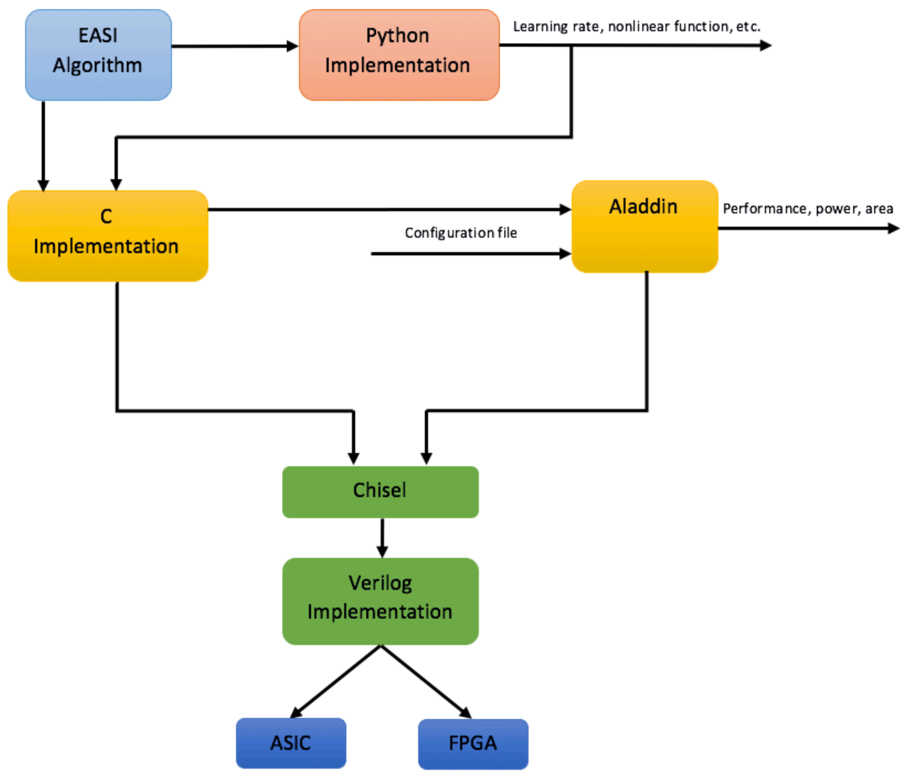


Figure 11: EASI implementation from software to hardware

Implementation in software allows us to quickly compare the quality of outputs for various nonlinear functions and learning rates. The Python implementation that will be turned in is a parameterized implementation of the EASI algorithm in the sense that the number of output dimensions can be determined by the user. After that, the learning rate can be tuned for the selected dimensions count. The nonlinear function used in this implementation is $g(h) = y^3$ and the learning rate is set to 0.001.

The basic building blocks required to implement the EASI algorithm include vector and matrix operations, such as vector-vector outer product, matrix-vector multiplication, and matrix-matrix multiplication/addition/subtraction. These operations can be implemented using nested loops in which various iterations of a loop are independent of each other. This introduces an opportunity for optimization where a loop may be unrolled fully or partially to increase parallelism at the cost of higher power and area consumption.

At a higher level of abstraction, EASI can be divided into macroblocks that have a limited communication with each other. This introduces another avenue for optimization which allows breaking down the hardware implementation into these macroblocks and use a pipelined architecture to increase throughput at the cost of slightly higher power and area consumption.

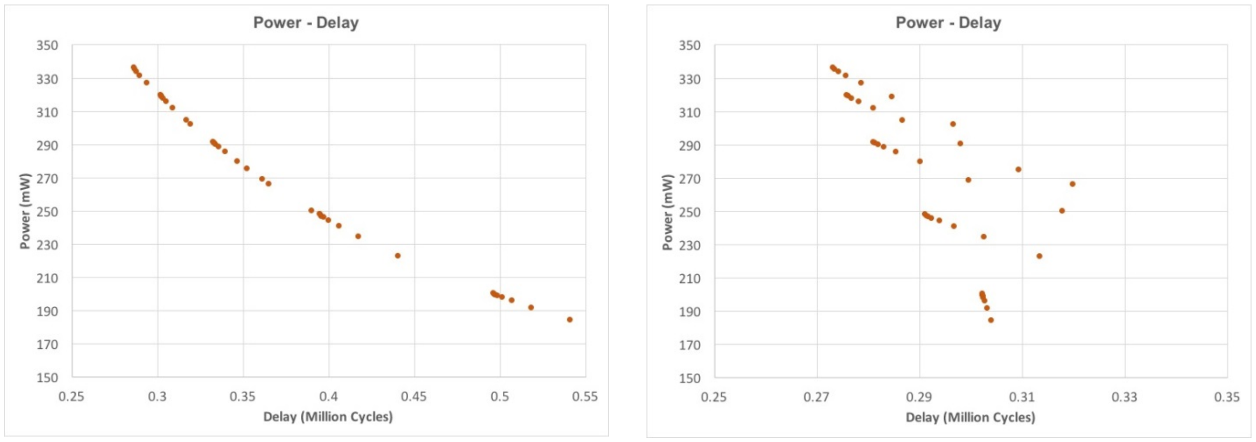


Figure 12: Design space exploration

Figure 12 shows the power-delay trade-offs for a non-pipelined design and a pipelined design (obtained using Aladdin [6]**)**. We can see that in a non-pipelined design (the left-hand plot), the power-delay curve is almost linear while in a pipelined design (the right-hand plot), there are regions where 5% increase in the delay can lead to 40% reduction in power consumption.

Baseline hardware implementation using Chisel

Because of the loop-carried dependency in the data dependency graph shown in Figure 9, EASI algorithm cannot be implemented efficiently using a pipelined architecture. As a result, we choose a single-cycle architecture to implement the algorithm. However, it may be possible to change the EASI algorithm to use a mini-batch gradient descent optimization instead of a stochastic gradient descent optimization to benefit from a pipelined architecture. This modification is explained later in the report.

All the operations shown in the data dependency graph of Figure 9 plus some additional operations including floating-point addition/subtraction/multiplication were implemented in Chisel. For each individual module, unit testing was performed and the modules passed all the tests. Additionally, the top module was tested and its correctness was verified.

Pipelined Implementation of the EASI algorithm

Because of the loop-carried data dependency in EASI with SGD, a fast and scalable hardware implementation is almost impossible. To design a hardware that is capable of operating at high speeds, we need to introduce a new optimization algorithm which is suitable for hardware implementation. Sequential mini-batch gradient descent (SMBGD) optimization is an update rule we propose to integrate with EASI. Similar to mini-batch gradient descent (MBGD) optimization, SMBGD allows multiple training samples to use the same separation matrix before updating the separation matrix for next iterations. As a result, a new training sample can be fed to the pipeline in each clock cycle, which improves the throughput. Additionally, it improves convergence by considering multiple training samples, in contrast to SGD that considers a single training sample at a time and causes noisier steps towards the minimum and may complicate convergence.

Our simulations show that using SMBGD can improve convergence rate by 24%, clock frequency by 3.2 times, and throughput by 13 times (for a 2-input, 2-output problem). The hardware resources required for implementing EASI with SMBGD are summarized as follows:

# of input dimensions	# of output dimensions	# of adaptive logic modules (ALMs)	# of registers	# of DSPs
2	2	6104	2144	26
4	2	10355	3377	42

4.2.3 VIBNN Hardware-Software Realization

The block diagram of the proposed hardware implementation of the VIBNN is shown in Figure 13. The ROM block stores software-learned variational parameters (for each connection in the VIBNN, there are two stored parameter values, one corresponding to the mean, the other to the variance.) The RAM is used to store intermediate results such as the layer output of intermediate layers. The core functionality of forward propagation is implemented using the layer computation block, which is comprised of multiple neuron computation blocks. Figure 14 shows the data-path in a neuron computation block, which calculates the output for one neuron. The inputs of the neuron (primary inputs or outputs from the previous layer) are read from the RAM block and multiplied with corresponding weights. The GRNG block produces a random value distributed per a zero-mean, unit-variance Gaussian distribution. This block, also called RLF-GRNG, was explained above. The Shift-and-Scale block takes as input the random number, which is generated by the GRNG block, and suitably shifts and scales this value so that the transformed value serves as a random number matching an arbitrary Gaussian distribution with the specified mean and variance, which were read from the ROM. The products are then accumulated at the next stage. Given the limited amount of resources, each neuron is updated in a time-multiplexed manner in which only a subset of its inputs and weights are multiplied and accumulated in each cycle. The activation function block is implemented as the rectified linear unit (ReLU) for its good performance as well as hardware complexity concern.

The full forward propagation in a BNN is implemented as a series of single-layer neuron computations. More precisely, a k-layer deep neural network is implemented in k steps, each step realizing exactly one of the k layers in increasing order. The layer computation block contains exactly eight neuron computation blocks, each of which implements the connection from eight input neurons to one output neuron. In total, the layer computation block can perform forward propagation from at most eight input neurons to at most eight output neurons simultaneously. If a layer has more than eight neurons and exceeds the bandwidth of the layer computation block, one has to time share the neuron computation blocks by decomposing the forward propagation into smaller problems. For example, the full forward propagation from 30 neurons to 30 neurons will be decomposed into $4 \times 4 = 16$ subproblems.

The controller coordinates memory access and arithmetic operations. The memory access addresses and the control signals are generated by the controller. The inputs, immediate layers outputs, and variational parameters are stored in the RAM and the ROM in ascending order. Therefore, memory accessing can be controlled by counter based logics. The controller implements three counters for read addresses for RAM, write addresses for RAM, and read addresses for ROM respectively. A finite state machine is implemented to generate all control signals such as register load and write enable according to the network structure.

Combining all aforementioned components, a VIBNN is implemented as a five-stage pipeline. The first two stages are used for random number generation. A pipeline stage is inserted after the weight update block. The last two stages are for the layer computation block for inner products calculations.

4.3 Evaluation of the VINE

We implement the proposed design of BNN on an Altera Cyclone V FPGA (Model Number 5CGTFD9E5F35C7) for the **MNIST dataset** [7]. Using a 784-200-200-10 network, a non-Bayesian ANN with Dropout applied can achieve 97.50% test accuracy. For a VIBNN, the software implementation can achieve 98.10% test accuracy whereas the hardware implementation can achieve 97.81% accuracy. In other words, the proposed implementation of BNN on FPGA degrades only 0.29% compared to its software model but can still achieve higher accuracy compared to a non-Bayesian ANN. The hardware resource utilization of the implemented BNN is shown in Table 3.

Table 3 Hardware utilization of the FPGA implementation of a BNN

Type of resource	**Utilization**
ALMs	2765/113,560 (2%)
Register count	3438
Block memory bits	4,355,456/12,492,800 (35%)
RAM blocks	426/1220 (35%)
DSP blocks	128/342 (37%)

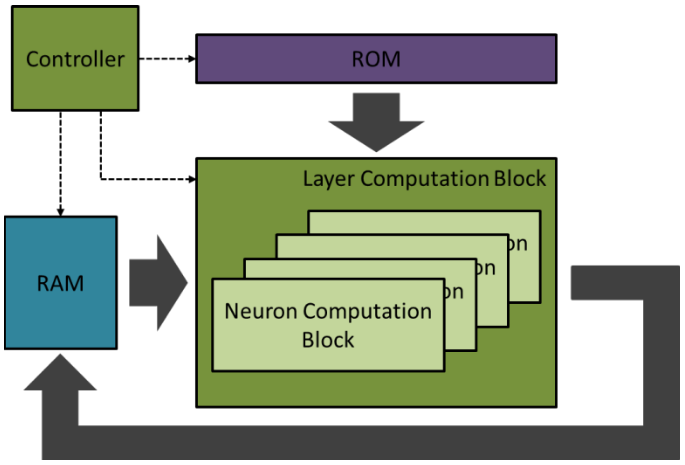


Figure 13: Block Diagram of a BNN

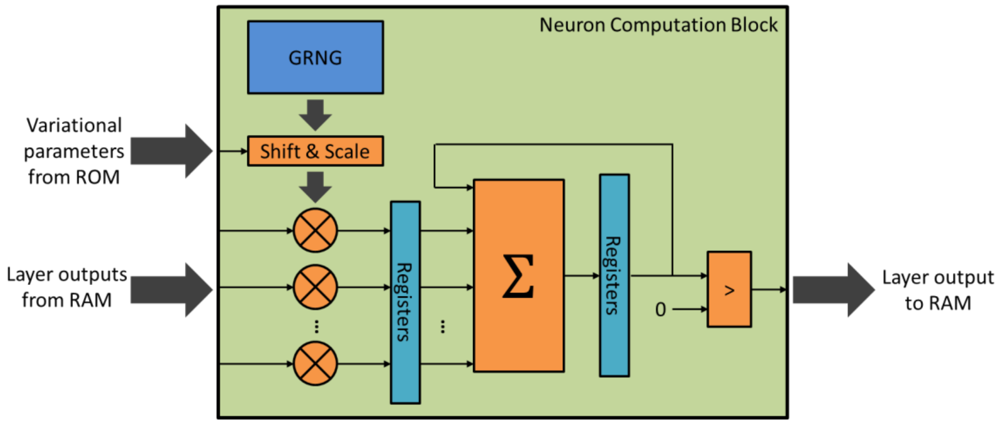


Figure 14: Datapath of a Neuron Computation Block

We have also applied the proposed framework to a mortality prediction task in which a VIBNN is used to predict the chance of survival of a number of patients given severity scores (such as SAPS-II [8]), prior clinical notes, and other information. We extracted from the MIMIC-III dataset [9] 91 features for each patient (gender, age, nine different severity scores, 20 Elixhauser comorbidity [10], and 50 features extracted from the clinical notes using a latent Dirichlet allocation [11]) and measured the area under receiver's operating characteristic curve (AUROC). A 91-256-256-1 network is used for this task. The software version without and quantization achieves an AUROC of 0.8479, which is similar to the result of ANN with Dropout (0.8493). The hardware version results in only a minor degradation and achieves an AUROC of 0.8457 while being significantly faster.

5. Conclusion

This report described our findings and results for the DARPA MTO seedling project titled "SpiNN-SC: Stochastic Computing-Based Realization of Spiking Neural Networks" also known as "VINE: A Variational Inference-Based Bayesian Neural Network Engine." A set of prototype software and hardware design deliverables for the said project was produced. Experimental results proved the effectiveness and anticipated benefits of the Bayesian Neural Network (BNN) with an integrated Variational Inference (VI) engine for performing inference and learning. Future work may apply the developed platform to a variety of applications ranging from natural language processing to cybersecurity, from dynamic energy governance in mobile systems to data center resource management, and from object classification to trend forecasting.

6. References

[1] C. Blundell, J. Cornebise, K. Kavukcuoglu and D. Wierstra, "Weight uncertainty in neural networks," *arXiv preprint arXiv:1505.05424,* 2015.

[2] G. E. Box, W. G. Hunter and J. S. Hunter, Statistics for experimenters: an introduction to design, data analysis, and model building, JSTOR, 1978.

[3] J.-F. Cardoso and B. H. Laheld, "Equivariant adaptive source separation," *IEEE Transactions on signal processing,* vol. 44, no. 12, pp. 3017-3030, 1996.

[4] J. Schlimmer, "Mushroom records drawn from The Audubon Society field guide to north American mushrooms," *GH Lincoff (Pres), New York,* 1981.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, 2012.

[6] Y. S. Shao, B. Reagen, G.-Y. Wei and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, 2014.

[7] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, no. 11, pp. 2278-2324, 1998.

[8] J.-R. Le Gall, S. Lemeshow and F. Saulnier, "A new simplified acute physiology score (SAPS II) based on a European/North American multicenter study," *Jama,* vol. 270, no. 24, pp. 2957-2963, 1993.

[9] A. E. Johnson, T. J. Pollard, L. Shen, L.-w. H. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. A. Celi and R. G. Mark, "MIMIC-III, a freely accessible critical care database," Nature Publishing Group, 2016.

[10] A. Elixhauser, C. Steiner, D. R. Harris and R. M. Coffey, "Comorbidity measures for use with administrative data," *Medical care,* vol. 36, no. 1, pp. 8-27, 1998.

[11] D. M. Blei, A. Y. Ng and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research,* vol. 3, no. Jan, pp. 993-1022, 2003.

7. List of Symbols, Abbreviations and Acronyms

ANN	Artificial Neural Network
BNN	Bayesian Neural Network
EASI	Equivariant Adaptive Separation via Independence
FPGA	Field Programmable Gate Array
GRNG	Gaussian Random Number Generator
HOS	High Order Statistics
ICA	Independent Component Analysis
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
MAP	Maximum A Posterior
MBGD	Mini-Batch Gradient Descent
MCMC	Markov Chain Monte Carlo
PC	Parallel Counter
PDF	Probability Density Function
RAM	Random Access Memory
RLF-GRNF	RAM-based Linear Feedback Gaussian Random Number Generator
ROM	Read Only Memory
RNG	Random Number Generator
SGD	Sequential Gradient Descent
SMBGD	Sequential Mini-Batch Gradient Descent
VI	Variational Inference
VIBNN	Variational Inference Based Bayesian Neural Network